Advanced Data Structures and Algorithms

Class Assignments

Name: Somyajit Jitendra S.

Class ID: A125022

Programme: M.Tech (Computer Science and Engineering)

Institute: International Institute of Information Technology, Bhubaneswar)

ADSA Assignment Questions

1. Prove that the time complexity of the recursive `Heapify` operation is $O(\log n)$ using the recurrence relation:
$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$
2. In an array of size n representing a binary heap, prove that all leaf nodes are located at indices from $\lfloor \frac{n}{2} \rfloor + 1$ to n.
3. (a) Show that in any heap containing n elements, the number of nodes at height h is at most:
$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$
   (b) Using the above result, prove that the time complexity of the `Build-Heap` algorithm is $O(n)$.
4. Explain the LU decomposition of a matrix using Gaussian Elimination. Clearly describe each step involved in the process.
5. Solve the following recurrence relation arising from the LUP decomposition solve procedure:
$$T(n) = \sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right] + \sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right]$$
6. Prove that if matrix A is non-singular, then its Schur complement is also non-singular.
7. Prove that positive-definite matrices are suitable for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.
8. For finding an augmenting path in a graph, should Breadth First Search (BFS) or Depth First Search (DFS) be applied? Justify your answer.
9. Explain why Dijkstra's algorithm cannot be applied to graphs with negative edge weights.
10. Prove that every connected component of the symmetric difference of two matchings in a graph G is either a path or an even-length cycle.
11. Define the class **Co-NP**. Explain the type of problems that belong to this complexity class.
12. Given a Boolean circuit instance whose output evaluates to `true`, explain how the correctness of the result can be verified in polynomial time using Depth First Search (DFS).
13. Is the **3-SAT (3-CNF-SAT)** problem NP-Hard? Justify your answer.
14. Is the **2-SAT** problem NP-Hard? Can it be solved in polynomial time? Explain your reasoning.

Contents

1 Time Complexity Analysis of the Recursive Heapify Operation

Problem Statement

Prove that the time complexity of the recursive `Heapify` operation is $O(\log n)$ using the recurrence relation:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Understanding the Heapify Operation

Heapify is a fundamental operation used in binary heaps to restore the heap property when it is violated at a node. It compares a node with its children and swaps it with the larger (or smaller, in a min-heap) child if required. This process is then applied recursively down the tree.

Key observations:

- Heapify operates on a **binary heap**, which is a complete binary tree.
- Each recursive call moves the node one level downward.
- At each level, only a constant number of comparisons and swaps are performed.

Given Recurrence Relation

The recurrence relation is:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

This means:

- The size of the problem reduces by a factor of at least $\frac{2}{3}$ at each recursive call.
- The non-recursive work done at each step is constant.

Solving the Recurrence

After k recursive calls, the problem size becomes:

$$n\left(\frac{2}{3}\right)^k$$

The recursion stops when the subproblem size becomes 1:

$$n\left(\frac{2}{3}\right)^k = 1$$

Taking logarithms:

$$k = \log_{3/2} n$$

Since logarithms with different bases differ only by a constant factor:

$$k = O(\log n)$$

Total Work Done

- Each recursive level performs $O(1)$ work
- Number of recursive levels is $O(\log n)$

Hence, total time complexity:

$$\boxed{T(n) = O(\log n)}$$

Conclusion

The recursive Heapify operation runs in logarithmic time because it traverses at most the height of the heap, which is $O(\log n)$ for a binary heap.

■

2 Position of Leaf Nodes in an Array-Based Binary Heap

Problem Statement

In an array of size n representing a binary heap, prove that all leaf nodes are located at indices:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n$$

Binary Heap Representation

A binary heap is stored as an array using level-order traversal.

For an element stored at index i:

- Parent index: $\lfloor i/2 \rfloor$
- Left child index: $2i$
- Right child index: $2i + 1$

Condition for a Node to be a Leaf

A node is a leaf if it has **no children**. Therefore, a node at index i is a leaf if:

$$2i > n$$

Solving:

$$i > \frac{n}{2}$$

Since array indices are integers:

$$i \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

Index Range of Leaf Nodes

Thus, all indices satisfying:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 \leq i \leq n$$

correspond to leaf nodes.

Why This Result Is Important

- Leaf nodes do not require heapify operations
- Build-Heap starts heapifying from index $\lfloor n/2 \rfloor$
- Helps optimize heap construction algorithms

Conclusion

All leaf nodes in a binary heap are stored from index $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n

■

3 Height-Based Analysis and Time Complexity of the Build-Heap Algorithm

Problem Statement

(a) Show that in a heap containing n elements, the number of nodes at height h is at most:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

(b) Using the above result, prove that the time complexity of the Build-Heap algorithm is $O(n)$.

3.1 (a) Number of Nodes at Height h

Understanding Heap Height

- Height of a node = number of edges on the longest path to a leaf
- Leaf nodes have height 0
- Height increases as we move upward toward the root

Node Distribution in a Heap

In a complete binary tree:

- The number of nodes halves as we move up each level
- Nodes at height h are located h levels above the leaves

Therefore, the maximum number of nodes at height h is:

$$\boxed{\left\lceil \frac{n}{2^{h+1}} \right\rceil}$$

3.2 (b) Time Complexity of Build-Heap

Common Misconception

Since Heapify takes $O(\log n)$ time, many incorrectly assume:

$$\text{Build-Heap} = O(n \log n)$$

This is incorrect.

Correct Analysis

- Nodes near the bottom have small height $\rightarrow$ cheap heapify
- Only a few nodes are near the root $\rightarrow$ expensive heapify

Total Cost Calculation

Total cost is:

$$\sum_{h=0}^{\log n} \left(\frac{n}{2^{h+1}} \cdot O(h) \right)$$

This summation converges to:

$$\boxed{O(n)}$$

Final Conclusion

$$\boxed{\text{The Build-Heap algorithm runs in linear time } O(n)}$$

■

4 LU Decomposition of a Matrix Using Gaussian Elimination

Problem Statement

Explain the LU decomposition of a matrix using Gaussian Elimination. Clearly describe each step involved in the process.

Introduction

LU decomposition is a fundamental technique in numerical linear algebra that factors a square matrix into the product of two triangular matrices. It plays a central role in solving systems of linear equations, matrix inversion, determinant computation, and numerical simulations.

The main idea is to decompose a given matrix:

$$A \in \mathbb{R}^{n \times n}$$

into:

$$\boxed{A = LU}$$

where:

- L is a **lower triangular matrix** with unit diagonal elements
- U is an **upper triangular matrix**

Motivation for LU Decomposition

Consider solving a linear system:

$$Ax = b$$

Using Gaussian elimination directly requires $O(n^3)$ operations. If we need to solve the system for multiple right-hand sides $b_1, b_2, \ldots, b_k$, repeating elimination each time is inefficient.

LU decomposition avoids this redundancy by:

- Performing elimination once: $A = LU$
- Solving $Ly = b$ using **forward substitution**
- Solving $Ux = y$ using **backward substitution**

Each substitution step takes only $O(n^2)$ time.

Gaussian Elimination as the Basis of LU

Gaussian elimination transforms a matrix into an upper triangular form by eliminating elements below the diagonal. The multipliers used during elimination naturally form the entries of matrix L.

Thus:

- Matrix U is the result of Gaussian elimination
- Matrix L stores elimination multipliers

Step-by-Step LU Decomposition

Consider the matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Step 1: Elimination in Column 1

To eliminate elements below a_{11}, compute multipliers:

$$l_{21} = \frac{a_{21}}{a_{11}}, \quad l_{31} = \frac{a_{31}}{a_{11}}$$

Apply row operations:

$$R_2 \leftarrow R_2 - l_{21}R_1, \quad R_3 \leftarrow R_3 - l_{31}R_1$$

The resulting matrix becomes:

$$U^{(1)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & u_{22} & u_{23} \\ 0 & u_{32} & u_{33} \end{bmatrix}$$

The multipliers are stored in L:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & 0 & 1 \end{bmatrix}$$

Step 2: Elimination in Column 2

To eliminate u_{32} using pivot u_{22}:

$$l_{32} = \frac{u_{32}}{u_{22}}$$

Apply:

$$R_3 \leftarrow R_3 - l_{32} R_2$$

Final upper triangular matrix:

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Update L:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}$$

Final Factorization

Thus:

$$\boxed{A = LU}$$

Conditions for Existence

- All pivot elements must be non-zero
- Pivoting may be required for numerical stability
- Positive-definite matrices do not require pivoting

Time and Space Complexity

- Time complexity: $O(n^3)$
- Space complexity: $O(n^2)$

Conclusion

LU decomposition transforms Gaussian elimination into a reusable factorization, making it a cornerstone of efficient numerical algorithms.

■

5 Solving the Recurrence Relation in the LUP Decomposition Solve Procedure

Problem Statement

Solve the recurrence relation arising from the LUP decomposition solve procedure:

$$T(n) = \sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right] + \sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right]$$

Context and Interpretation

This recurrence models the computational cost of:

- **Forward substitution** while solving $Ly = b$
- **Backward substitution** while solving $Ux = y$

  Each phase consists of nested loops over matrix indices.

Analysis of the First Summation

$$\sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right]$$

Inner Loop

For a fixed i, the inner loop executes $i - 1$ times:

$$\sum_{j=1}^{i-1} O(1) = O(i)$$

Outer Loop

Substituting:

$$\sum_{i=1}^{n} O(i) = \frac{n(n+1)}{2} = O(n^2)$$

Thus, the first summation contributes $O(n^2)$.

Analysis of the Second Summation

$$\sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right]$$

Inner Loop

For fixed i, the inner loop runs $n - i$ times:

$$\sum_{j=i+1}^{n} O(1) = O(n - i)$$

Outer Loop

$$\sum_{i=1}^{n} O(n - i) = \frac{n(n-1)}{2} = O(n^2)$$

Total Time Complexity

Combining both parts:

$$T(n) = O(n^2) + O(n^2) = \boxed{O(n^2)}$$

Significance of the Result

- LU decomposition costs $O(n^3)$
- Solving each system after decomposition costs only $O(n^2)$
- This explains the efficiency of LU/LUP-based solvers

Conclusion

The recurrence confirms that forward and backward substitution together run in quadratic time, validating the efficiency of LUP solve procedures.

■

6 Non-Singularity of the Schur Complement of a Matrix

Problem Statement

Prove that if a matrix A is non-singular, then its Schur complement (with respect to an invertible block) is also non-singular.

Introduction

The Schur complement is a central concept in matrix theory and numerical linear algebra. It appears naturally in:

- Block LU decomposition
- Recursive matrix algorithms
- Optimization problems
- Solving large linear systems using divide-and-conquer strategies

Understanding its non-singularity properties is crucial for ensuring the correctness and stability of such algorithms.

Block Partitioning of the Matrix

Let the matrix A be partitioned as:

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

where:

- $B \in \mathbb{R}^{k \times k}$ is a square, invertible matrix
- $C \in \mathbb{R}^{k \times (n-k)}$
- $D \in \mathbb{R}^{(n-k) \times k}$
- $E \in \mathbb{R}^{(n-k) \times (n-k)}$

Definition of the Schur Complement

The **Schur complement** of block B in matrix A is defined as:

$$\boxed{S = E - DB^{-1}C}$$

Intuitively, S represents the remaining system after eliminating the effect of block B.

Goal of the Proof

We want to show:

$$A \text{ is non-singular } \Rightarrow S \text{ is non-singular}$$

The proof will use:

- Block matrix factorization
- Determinant properties

Block LU-Type Factorization

Since B is invertible, matrix A can be factored as:

$$A = \begin{bmatrix} I & 0 \\ DB^{-1} & I \end{bmatrix} \begin{bmatrix} B & C \\ 0 & E - DB^{-1}C \end{bmatrix}$$

Explanation of the Factorization

- The first matrix is a lower triangular block matrix with identity blocks
- The second matrix is an upper triangular block matrix
- Their product exactly reconstructs A

Determinant Analysis

Taking determinants on both sides:

$$\det(A) = \det \begin{bmatrix} I & 0 \\ DB^{-1} & I \end{bmatrix} \cdot \det \begin{bmatrix} B & C \\ 0 & S \end{bmatrix}$$

Using determinant properties:

- Determinant of a triangular block matrix equals the product of diagonal blocks
- $\det(I) = 1$

Thus:

$$\det(A) = \det(B) \cdot \det(S)$$

Non-Singularity Argument

- Given: A is non-singular $\Rightarrow \det(A) \neq 0$
- B is invertible $\Rightarrow \det(B) \neq 0$

Hence:

$$\det(S) \neq 0$$

Conclusion

If matrix A is non-singular, then its Schur complement S is also non-singular.

■

7 LU Decomposition of Positive-Definite Matrices without Pivoting

Problem Statement

Prove that positive-definite matrices are suitable for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.

Introduction

In general, LU decomposition may fail if a pivot element becomes zero or very small. To handle such cases, **pivoting** is introduced.

However, for an important class of matrices known as **positive-definite matrices**, LU decomposition can be safely performed *without* pivoting. This property is crucial in scientific computing and optimization.

Definition of Positive-Definite Matrix

A real symmetric matrix $A \in \mathbb{R}^{n \times n}$ is said to be **positive definite** if:

$$x^T Ax > 0 \quad \forall x \neq 0$$

Key Properties of Positive-Definite Matrices

Positive-definite matrices satisfy the following important properties:

- All eigenvalues of A are strictly positive
- All leading principal minors of A are positive
- A is non-singular

These properties are fundamental to the proof.

Pivot Elements in LU Decomposition

In Gaussian elimination:

- Pivot elements are the diagonal entries of matrix U
- Each pivot appears in the denominator when computing multipliers

If any pivot is zero, the algorithm fails due to division by zero.

Why Positive-Definite Matrices Avoid This Issue

For a positive-definite matrix:

$$\det(A_k) > 0 \quad \forall k = 1, 2, \ldots, n$$

where A_k is the $k \times k$ leading principal submatrix.

Connection to Pivot Elements

Each pivot element in LU decomposition can be expressed as:

$$u_{kk} = \frac{\det(A_k)}{\det(A_{k-1})}$$

Since both determinants are positive:

$$u_{kk} > 0$$

Thus:

- All pivots are non-zero
- No division by zero occurs

Recursive LU and Schur Complements

In recursive LU decomposition:

- The matrix is partitioned into blocks
- Schur complements are computed at each recursion step

Positive-definiteness is preserved under Schur complementation. Hence, each recursive subproblem:

- Remains positive definite
- Has non-zero pivots

Comparison with General Matrices

Matrix Type	Pivoting Required	Reason
General matrix	Yes	Zero or unstable pivots possible
Positive-definite matrix	No	Pivots always strictly positive

Conclusion

Positive-definite matrices guarantee non-zero pivots, therefore LU decomposition can be performed safely without pivoting. This property makes positive-definite matrices especially important in numerical linear algebra, optimization, and scientific computing.

■

8 Choice of BFS or DFS for Finding Augmenting Paths

Problem Statement

For finding an augmenting path in a graph, should Breadth First Search (BFS) or Depth First Search (DFS) be applied? Justify your answer.

Introduction

Augmenting paths are a fundamental concept in graph algorithms, particularly in:

- Maximum flow algorithms
- Bipartite matching problems
- Network optimization

An augmenting path is a path along which the current solution (flow or matching) can be improved.

The choice of graph traversal strategy directly affects the efficiency and correctness of the algorithm.

Understanding Augmenting Paths

An augmenting path is defined as:

- A path from the source to the sink in a residual graph (flow problems), or
- A path that alternates between unmatched and matched edges (matching problems)

Using an augmenting path allows the algorithm to:

- Increase the total flow, or
- Increase the size of the matching

Breadth First Search (BFS)

BFS explores the graph level by level. When applied to augmenting path problems:

- BFS finds the **shortest augmenting path** in terms of number of edges
- This minimizes the number of augmentation steps required

This idea is central to:

- Edmonds–Karp algorithm (for max flow)
- Hopcroft–Karp algorithm (for bipartite matching)

Depth First Search (DFS)

DFS explores paths deeply before backtracking. While DFS can find augmenting paths:

- The paths found may be unnecessarily long
- Long augmenting paths lead to inefficient progress
- The number of augmentations may increase significantly

In worst cases, DFS-based augmentation can degrade performance severely.

Why BFS is Preferred

- Shorter augmenting paths reduce the number of iterations
- BFS ensures polynomial-time guarantees in algorithms like Edmonds–Karp
- BFS avoids pathological cases where DFS repeatedly chooses poor paths

Conclusion

Breadth First Search is preferred over Depth First Search for finding augmenting paths because it guarantees shorter paths, faster convergence, and better theoretical time complexity bounds in flow and matching algorithms.

■

9 Failure of Dijkstra's Algorithm in the Presence of Negative Edge Weights

Problem Statement

Explain why Dijkstra's algorithm cannot be applied to graphs that contain negative edge weights.

Introduction

Dijkstra's algorithm is a classical shortest path algorithm used to compute the minimum distance from a source vertex to all other vertices in a weighted graph.

However, it is well known that Dijkstra's algorithm fails in the presence of negative edge weights. Understanding the reason for this failure is crucial for correct algorithm selection.

Key Assumption of Dijkstra's Algorithm

Dijkstra's algorithm relies on the following fundamental assumption:

> Once a vertex is extracted from the priority queue, its shortest distance from the source is final and will never change.

This assumption holds only when all edge weights are non-negative.

How the Algorithm Works

At each step:

- The algorithm selects the unvisited vertex with minimum tentative distance
- It relaxes all outgoing edges from that vertex
- The selected vertex is permanently labeled

This greedy choice is valid only if no future relaxation can reduce the distance of a finalized vertex.

Effect of Negative Edge Weights

When negative edge weights exist:

- A path discovered later may reduce the distance of an already finalized vertex
- This directly violates Dijkstra's core assumption

Thus, the algorithm may commit to incorrect distances prematurely.

Conceptual Counterexample

Consider a situation where:

- Vertex u is finalized with distance $d(u)$
- A different path through another vertex leads to u with smaller total cost due to a negative edge

Since u is already finalized, Dijkstra's algorithm cannot update its distance, leading to incorrect results.

Correct Alternatives

Graphs with negative edge weights require algorithms such as:

- Bellman–Ford algorithm
- Johnson's algorithm (for all-pairs shortest paths)

These algorithms allow distance updates even after multiple relaxations.

Conclusion

Dijkstra's algorithm fails with negative edge weights because its greedy strategy assumes that distances only increase, an assumption that is violated when negative edges are present.

■

10 Structure of the Symmetric Difference of Two Matchings

Problem Statement

Prove that every connected component of the symmetric difference of two matchings in a graph G is either a path or an even-length cycle.

Introduction

Matchings are a central concept in graph theory and play a key role in:

- Bipartite matching algorithms
- Network flow theory
- Combinatorial optimization

Analyzing the structure of the symmetric difference of matchings is essential for understanding augmenting path techniques.

Basic Definitions

Let $G = (V, E)$ be an undirected graph.

- A **matching** is a set of edges such that no two edges share a common vertex
- Let M_1 and M_2 be two matchings in G
- The **symmetric difference** is defined as:

$$M_1 \oplus M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$$

Degree Property of the Symmetric Difference

Since M_1 and M_2 are matchings:

- Each vertex is incident to at most one edge from M_1
- Each vertex is incident to at most one edge from M_2

Therefore, in the graph formed by $M_1 \oplus M_2$:

$$\text{degree of every vertex } \leq 2$$

This observation severely restricts the possible structures of connected components.

Analysis of Connected Components

Let C be a connected component of $M_1 \oplus M_2$.

Since every vertex in C has degree at most 2, the component can only take one of the following forms:

- A path
- A cycle

No other structure is possible.

Alternating Edge Property

Edges in $M_1 \oplus M_2$ must alternate between:

- Edges from M_1
- Edges from M_2

This alternation is forced by the definition of matchings.

Why Cycles Must Be Even-Length

In a cycle:

- Edges must alternate between M_1 and M_2
- Alternation requires pairs of edges

An odd-length cycle would force two adjacent edges to belong to the same matching, which contradicts the definition of a matching.

Final Argument

- Each connected component has maximum degree 2
- Hence, it is a path or a cycle
- Cycles must be even-length due to edge alternation

Conclusion

Every connected component of the symmetric difference of two matchings in a graph is either a path or an even-length cycle, a property fundamental to matching and flow algorithms.

■

11 Definition and Characteristics of the Complexity Class Co-NP

Problem Statement

Define the complexity class Co-NP. Explain the type of decision problems that belong to this class.

Introduction

In computational complexity theory, decision problems are classified based on the difficulty of verifying their solutions. One of the most fundamental complexity classes in this context is NP. Closely related to NP is another important class called Co-NP.

Understanding Co-NP is essential for analyzing the structure of complexity classes and the limits of efficient computation.

Background: Class NP

A decision problem belongs to the class NP if:

- The answer is **YES**
- There exists a certificate (also called a witness)
- The certificate can be verified in polynomial time

In other words, NP contains problems for which a proposed solution can be efficiently checked.

Definition of Co-NP

A decision problem belongs to the class Co-NP if:

- Its complement problem belongs to NP

Equivalently, a problem is in Co-NP if:

- The answer is **NO**
- There exists a certificate for the NO answer
- The certificate can be verifiable in polynomial time

Thus, Co-NP focuses on efficiently verifiable NO instances.

Intuitive Interpretation

The distinction between NP and Co-NP can be summarized as follows:

- NP: Easy to verify YES answers
- Co-NP: Easy to verify NO answers

This asymmetry is fundamental to many open questions in complexity theory.

Examples of Problems in Co-NP

Some well-known problems that belong to Co-NP include:

- **UNSAT**: Given a Boolean formula, prove that it is unsatisfiable
- **TAUTOLOGY**: Determine whether a Boolean formula is true for all assignments
- **Composite Number**: Prove that a given integer is not prime

In each case, a NO certificate can be verified efficiently.

Relationship Between NP and Co-NP

It is known that:

$$P \subseteq NP \cap Co\text{-}NP$$

However, it is still an open problem whether:

$$NP = Co\text{-}NP$$

Resolving this question would have deep consequences for theoretical computer science.

Conclusion

Co-NP captures decision problems for which incorrectness can be efficiently verified, making it a central concept in computational complexity theory.

■

12 Polynomial-Time Verification of Boolean Circuit Outputs Using DFS

Problem Statement

Given a Boolean circuit instance whose output evaluates to TRUE, explain how the correctness of the result can be verified in polynomial time using Depth First Search (DFS).

Introduction

Boolean circuits are widely used to represent computations in complexity theory and digital logic. Verifying the correctness of a circuit's output is an important task related to NP verification.

Rather than recomputing the entire circuit from scratch, we aim to verify the correctness efficiently.

Boolean Circuit as a Graph

A Boolean circuit can be modeled as a directed acyclic graph (DAG), where:

- Vertices represent logic gates (AND, OR, NOT)
- Directed edges represent signal flow
- Input nodes have fixed Boolean values
- The output node produces the final result

Since the circuit is acyclic, DFS traversal is well-defined.

Key Idea of Verification

Given that the output gate evaluates to TRUE:

- We verify whether each gate is logically consistent
- We do not attempt to find a satisfying assignment

- We only check correctness of the claimed output

Role of Depth First Search

DFS is applied starting from the output gate:

- Traverse backward through the circuit
- Recursively visit all input gates influencing the output
- Verify gate correctness in a bottom-up manner

Gate-Level Verification

During DFS, each gate is checked according to its type:

- AND gate: Output TRUE only if all inputs are TRUE
- OR gate: Output TRUE if at least one input is TRUE
- NOT gate: Output TRUE if input is FALSE

If any gate violates its logical condition, the output is declared incorrect.

Time Complexity Analysis

- Each gate is visited exactly once
- Each wire (edge) is examined once

Hence, total verification time is:

$$O(V + E)$$

Since the size of the circuit is polynomial in the input size, verification runs in polynomial time.

Conclusion

DFS provides an efficient mechanism for verifying Boolean circuit outputs, establishing that Boolean circuit evaluation belongs to NP.

■

13 NP-Hardness of the 3-SAT (3-CNF-SAT) Problem

Problem Statement

Is the 3-SAT (3-CNF-SAT) problem NP-Hard? Justify your answer.

Introduction

The Boolean satisfiability problem (SAT) was the first problem proven to be NP-Complete. 3-SAT is a restricted version of SAT where each clause contains exactly three literals.

Despite this restriction, 3-SAT remains computationally difficult.

Definition of 3-SAT

In the 3-SAT problem:

- The Boolean formula is expressed in Conjunctive Normal Form (CNF)
- Each clause contains exactly three literals

An example of a 3-SAT formula is:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_5)$$

Understanding NP-Hardness

A problem is NP-Hard if:

- Every problem in NP can be reduced to it
- The reduction runs in polynomial time

To prove NP-Hardness of 3-SAT, it suffices to show a polynomial-time reduction from SAT to 3-SAT.

Reduction from SAT to 3-SAT

It has been shown that:

- Any Boolean formula in CNF can be transformed into an equivalent 3-CNF formula
- The transformation preserves satisfiability
- The size of the formula increases only polynomially

This means that:

- The original SAT instance is satisfiable if and only if the transformed 3-SAT instance is satisfiable

Implication of the Reduction

Since:

- SAT is NP-Complete
- SAT $\leq_p$ 3-SAT

It follows that 3-SAT is NP-Hard.

Membership in NP

In addition:

- Given a truth assignment, the satisfaction of a 3-SAT formula can be verified in polynomial time

Thus, 3-SAT belongs to NP as well.

Conclusion

3-SAT is NP-Hard due to the existence of a polynomial-time reduction from SAT, and since it also belongs to NP, it is NP-Complete.

■

14 Polynomial-Time Solvability and Complexity of the 2-SAT Problem

Problem Statement

Is the 2-SAT problem NP-Hard? Can it be solved in polynomial time? Explain your reasoning.

Introduction

The Boolean satisfiability problem has several restricted variants that differ significantly in computational complexity. One of the most important such variants is the **2-SAT problem**.

Unlike 3-SAT, which is NP-Complete, 2-SAT admits efficient polynomial-time algorithms. Understanding this distinction highlights a fundamental boundary in the theory of computational complexity.

Definition of 2-SAT

In the 2-SAT problem:

- The Boolean formula is expressed in Conjunctive Normal Form (CNF)
- Each clause contains at most two literals

An example of a 2-SAT formula is:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$$

Comparison with 3-SAT

Although 2-SAT and 3-SAT appear syntactically similar, their computational complexities differ dramatically. Restricting each clause to at most two literals removes the combinatorial explosion present in 3-SAT.

Why 2-SAT is Not NP-Hard

A problem that can be solved in polynomial time cannot be NP-Hard unless $P = NP$. Since 2-SAT has known polynomial-time algorithms, it follows that 2-SAT is not NP-Hard unless the widely believed conjecture $P \neq NP$ is false.

Implication Graph Representation

The key insight behind polynomial-time solvability of 2-SAT is its graph-based interpretation.

Each clause $(a \vee b)$ can be rewritten as:

$$(\neg a \Rightarrow b) \quad \text{and} \quad (\neg b \Rightarrow a)$$

Using this transformation:

- Each variable and its negation become vertices
- Each implication becomes a directed edge

The resulting structure is called the **implication graph**.

Key Correctness Condition

A 2-SAT formula is satisfiable if and only if:

- No variable x and its negation $\neg x$ belong to the same strongly connected component (SCC)

If such a situation occurs, it would imply both $x \Rightarrow \neg x$ and $\neg x \Rightarrow x$, leading to a logical contradiction.

Polynomial-Time Algorithm

The standard algorithm for solving 2-SAT proceeds as follows:

1. Construct the implication graph
2. Compute strongly connected components using:
    - Kosaraju's algorithm, or
    - Tarjan's algorithm
3. Check for each variable whether x and $\neg x$ lie in the same SCC

Time Complexity Analysis

Let:

- $V = O(n)$ be the number of vertices
- $E = O(m)$ be the number of edges

Computing SCCs takes:

$$O(V + E)$$

which is polynomial in the input size.

Conclusion

2-SAT admits an efficient polynomial-time solution through graph-based techniques. Therefore, it is not NP-Hard and stands in sharp contrast to the NP-Complete 3-SAT problem, demonstrating how small syntactic restrictions can dramatically change computational complexity.

■